Compilation of Linearizable Data Structures
A Mechanised RG Logic for Semantic Refinement

Yannick Zakowski[1], David Cachera[1], Delphine Demange[2⋆], and David Pichardie[1]

[1] IRISA / ENS Rennes / Inria
[2] IRISA / University of Rennes 1 / Inria

Abstract. Modern programming languages provide libraries for concurrent data structures. For better performance, these are implemented with fine-grained concurrency. Still, such implementations are *linearizable*: the programmer can safely assume that they behave atomically.
We formalize this insight in Coq as an end-to-end theorem establishing the semantic preservation of a compiler translating abstract, atomic data structures into their concrete, fine-grained concurrent implementation. This embeds the notion of linearizable data structures in a formally verified compiler.
At the crux of the proof lies a generic result establishing, once and for all, a simulation relation, starting from a carefully crafted rely-guarantee specification. Inspired by the work of Vafeiadis, implementations are annotated with linearization points, which instrument programs semantics to reflect the behavior of abstract data structures. We successfully applied our generic theorem to concurrent buffers, a data structure used in the implementation of concurrent garbage collectors.

Keywords: Verified Compilation, Linearizability, Rely Guarantee, Coq

1 Introduction

Modern programming languages like Java or C++ provide rich libraries with efficient services and data structures, *e.g.* stacks, queues or sets. In a concurrent setting, efficiency is often achieved by means of *fine-grained concurrency*, where synchronization costs are reduced to a minimum. These algorithms are extremely subtle, can contain data races, and are hard to implement correctly.

Experts who program such algorithms generally resort on well chosen data structures that can be accessed using *linearizable* methods: even though the implementation of these methods is by no means atomic, they appear to the rest of the system to occur instantaneously. Linearizability thus considerably simplifies the understanding: the programmer can safely reason at a higher-level, and assume an abstract, atomic specification for these data structures.

Initially, linearizability was formally defined by Herlihy and Wing [7]. In their seminal paper, they model systems as I/O automata producing event

⋆ This work was supported by Agence Nationale de la Recherche, grant number ANR-14-CE28-0004 DISCOVER.

traces (histories), and a system is linearizable whenever all valid histories can be reordered into sequential histories.

In this work, we rather consider an alternative formulation, based on semantic refinement. Indeed, we prove an end-to-end compiler correctness theorem between a source language featuring atomic data-structures, and a target language where data-structures are implemented with fine-grained concurrency. In a nutshell, we want to establish that, for any source program p, $\texttt{obs}(\texttt{compile}(p)) \subseteq \texttt{obs}(p)$, where `obs` denotes the observable behaviors of a program. The reasons for this choice are twofold. First, compiler verification has now become widespread, and an accessible concept, even for non-expert. Second, this result serves as an intermediate layer in the verification of complex language runtime artifacts, such as garbage collectors or memory allocators: their correctness can be proved more easily when reasoning at a higher level on the data structures they manipulate.

Proving the above theorem is usually done by showing that the target program $\texttt{compile}(p)$ simulates the source program p: for any execution of the target program, we must exhibit a matching execution of the source program. While the definition of the matching relation is relatively intuitive, proving that it is indeed maintained along the execution can be cumbersome. Hence, we would like to resort to popular program verification techniques.

In his PhD, Vafeiadis [16] proposed a promising approach that fits our needs. It is based on Rely-Guarantee (RG) [9], a popular proof technique extending Hoare logic to concurrency. In RG, interferences between threads are described by binary relations on shared states. Each thread is proved correct under the assumption that other threads obey a *rely* relation. The effect of the thread itself must respect a *guarantee* relation, which must be accounted for in the relies of the other threads. RG allows for thread-modular reasoning, and hence removes the need to explicitly consider all interleavings in a global fashion. Now, RG alone does not capture the notion of linearizable method. So Vafeiadis proposes to extend RG to *hybrid* implementations, *i.e.* fine-grained concurrent methods instrumented with linearization points that reflect the abstract, atomic behavior of methods in a ghost part of the execution state. The approach is elegant, but is not formally linked to the above standard notion of compiler correctness. Recently, Liang et al. [12] improved on the work of Vafeiadis by expressing the soundness of the methodology with a semantic refinement. Their work undoubtedly makes progress in the right direction. Unfortunately, their proof is not machine-checked.

To sum up, we provide a machine-checked semantic foundation to the approach outlined in Vafeiadis' PhD, and embed it in a generic, end-to-end compiler correctness theorem. More precisely, we make the following contributions:

- We integrate the notion of linearizable data structures in a formally verified compiler. Correctness is phrased in terms of a simple semantic refinement and avoids the difficult, though traditional, definition of linearizability.
- Our proof is generic in the abstract data structures under consideration, and in the source program using them. The underlying simulation is proved once and for all, provided that hybrid implementations meet a certain specification.

– We express this specification in terms of RG reasoning, so it integrates smoothly with deductive proof systems.
– All theorems are proved correct in Coq, and available online [19].

2 Challenges and Overview

At the source level, our core concurrent language $\mathcal{L}^\sharp$ features *abstract data structures* with a set $\mathcal{I}$ of atomic methods. We program a compiler $\texttt{compile}\langle\mathcal{I}\rangle \in \mathcal{L}^\sharp \to \mathcal{L}$, which replaces the abstract data structures with their fine-grained concurrent implementation in $\mathcal{L}$. Our goal is to prove that this compiler is correct, in the sense that it preserves the observable behaviors of source programs, with a theorem of the form $\forall p \in \mathcal{L}^\sharp,\ \texttt{obs}(\texttt{compile}\langle\mathcal{I}\rangle(p)) \subseteq \texttt{obs}(p)$.

Our main result is generic in the abstract data structures used in $\mathcal{L}^\sharp$ and their implementation $\mathcal{I}$. Here, we illustrate on a simple example what are the intrinsic challenges, and briefly overview our technical contribution.

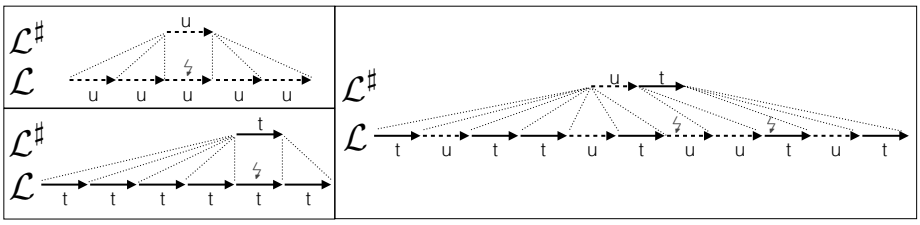
Running Example. Suppose $\mathcal{L}^\sharp$ offers abstract locks, with methods $\mathcal{I} = \{\texttt{acquire}, \texttt{release}\}$. At the level of $\mathcal{L}^\sharp$, an abstract lock can be seen as a simple boolean value, with the expected atomic semantics. At the concrete level, $\mathcal{L}$, `acquire` and `release` are implemented with the code in Figure 1. They use a boolean field `flag`, denoting the status of the lock. Releasing the lock simply sets the field to 0, while acquiring it requires a *compare-and-swap* instruction (`cas`) to guarantee mutual exclusion. Note that both methods could be executed concurrently by two client threads.

```
def acquire() ::=
  ok = 0
  do {
   ok=cas(this.flag,0,1)
  } while (ok == 0)
  return

def release() ::=
  this.flag = 0
  return
```

Fig. 1: Spinlock in $\mathcal{L}$

Semantic Refinement. Proving the above theorem is done with a simulation: for any execution of the target program, we must exhibit a matching execution of the source program. Between $\mathcal{L}$ and $\mathcal{L}^\sharp$, the simulation is particularly difficult to establish. The figure below illustrates the problem. Execution steps labelled with $\lightning$ are those where the effect of a method in $\mathcal{I}$ becomes visible to other threads, and determines the behavior of other methods in $\mathcal{I}$.



At the intra-thread level (left), we need to relate several steps of a thread in the target program to a single step in the source. The situation is even more difficult at the inter-thread level (right): the matching scheduling of threads `t` and `u` at

the source level is not necessarily the same as the one at the target level. It all depends on which thread will be the first to execute its $\lightning$ step in the concrete execution. The matching step for a given thread hence depends on the execution of its environment.

Our main result removes this difficulty by establishing, under some hypotheses, a *generic* simulation that entails semantic refinement. This is a meta-theorem that we establish once and for all, independently of the abstract data-structures available in $\mathcal{L}^\sharp$.

Rely-Guarantee for Atomicity. To prove the atomicity of linearizable methods, and establishing the simulation, we introduce an intermediate, proof-dedicated language $\mathcal{L}^\flat$, that comes with an RG proof system $\mathcal{L}^\flat$ provides *explicit linearization point* annotations, $\texttt{Lin}(b)$, to guide the proof. $\texttt{Lin}(b)$ annotations have an operational effect: they trigger, whenever condition b is met, the execution of the abstract method in a ghost part of the state. They are what makes $\mathcal{L}^\flat$ *hybrid*. Additionally, they allow to track whether the thread has reached the linearization point or not, thus helping the construction of our generic simulation. The annotated spinlock code is shown in Figure 2.

```
def acquire() ::=
  ok = 0 ;
  do {
    atomic ⟨
      ok=cas(this.flag,0,1)
      Lin(ok==1)⟩
  } while (ok == 0)
  return

def release() ::=
  atomic ⟨
    this.flag = 0
    Lin(true)⟩
  return
```

Fig. 2: Spinlock in $\mathcal{L}^\flat$

Hence, our compiler, defined as $\texttt{compile} = \texttt{clean} \circ \texttt{concretize}$, first transforms a source program p into $\texttt{concretize}\langle\mathcal{I}\rangle(p) \in \mathcal{L}^\flat$ where abstract methods are implemented by hybrid, annotated code. Then, the second compilation phase $\texttt{clean}\langle\mathcal{I}\rangle \in \mathcal{L}^\flat \to \mathcal{L}$ takes care of removing `Lin` instructions in the target program. We prove in Coq that the compiler is correct, providing that hybrid methods in $\mathcal{I}$ are proved correct *w.r.t.* an RG specification, RGspec, that we carefully define, in terms of $\mathcal{L}^\flat$ semantics, to prove, via the aforementioned simulation:

$$\mathsf{RGspec}\,(\mathcal{I}) \implies \forall p \in \mathcal{L}^\sharp,\ \texttt{obs}(\texttt{compile}\langle\mathcal{I}\rangle(p)) \subseteq \texttt{obs}(p)$$

Cleaning the Proof Instrumentation. We also embed in judgment RGspec the requirements sufficient to show that `Lin` instructions can be safely removed by the `clean` phase. This ensures that, despite their operational nature, `Lin` instructions are only passively intrumenting the program and its semantics.

Using our Result. The typical workflow for using our generic result is to (i) define the abstract data structures specification, *i.e.* their type, and the atomic semantics of methods in $\mathcal{I}$, (ii) provide a concrete implementation, *i.e.* their representation in the heap and a fine-grained hybrid implementation of methods, (iii) define a coherence invariant between abstract and concrete data structures (iv) define the rely and guarantee of each method, (v) prove the RG specification of each method using a dedicated program logic[3] and (vi) apply our meta-theorem to get

[3] This contribution is out of the scope of this paper, though we use it in our development.

$\langle expr \rangle$ e ::= n | null | x | e + e | - e | e mod n | ...
$\langle bexpr \rangle$ b ::= true | e == e | e <> e | b && b | b || b | !b | ...
$\langle comm \rangle$ c ::= • | assume(b) | print(e) | x = e | x = y.f | x.f = y | x = new(f,...,f)
| return(e) | x = y.m(z) | c ; c | c + c | loop(c) | atomic ⟨c⟩
$\langle comm \rangle^\sharp$ $c^\sharp$ ::= c | x =# y.m(z)
$\langle comm \rangle^\flat$ $c^\flat$ ::= c | Lin (b)

Fig. 3: Language Syntax

the global correctness result. We have successfully used this workflow to prove the correctness of the above spinlock example, as well as concurrent buffers, a data structure used in the implementation of a concurrent garbage collector [2,18].

3 The Languages and their Semantics

As explained in the previous section, this work considers three different languages $\mathcal{L}^\sharp$, $\mathcal{L}^\flat$ and $\mathcal{L}$. To lighten the presentation, however, we will just assume one language, $\mathcal{L}^\flat$, that includes all features, and keep in mind that source programs in $\mathcal{L}^\sharp$ do not include any linearization instrumentation, while target programs in $\mathcal{L}$ do not contain abstract method calls nor linearization instrumentation.

$\mathcal{L}^\sharp$ is a concurrent imperative language, with no dynamic creation of threads. It is dynamically typed, and features a simplified object model: objects in the heap are just records, and rather than virtual method calls, the current object – the object whose method is being called – is an extra function argument, passed in the reserved variable `this`. In the sequel, *Var* is a set of variables identifiers, method names range over $\mathtt{m} \in \mathit{Methods}$, and fields identifiers range over $\mathtt{f} \in \mathit{Fields}$.

Values and Abstract Data Structures We use the domain of values $\mathit{Val} = \mathbb{Z} + \mathit{Ref} + \mathit{Null}$, where *Ref* is a countable set of references. A central notion in the language is the one of abstract data structure. They are specified with an atomic specification. All our development and our proofs are parameterized by an abstract data structure specification. It could be abstract locks as in Section 2, bags, stacks, or buffers as detailed in Section 5.

Definition 1 (Abstract data structure specification). *An abstract data structure is specified by a tuple* $(A^\sharp, \mathcal{I}, [\![.]\!]^\sharp, \mathcal{P})$ *where* $A^\sharp$ *is a set of abstract objects;* $\mathcal{I} \subseteq \mathit{Methods}$ *is a set of abstract methods identifiers, whose atomic semantics is given by the partial map* $[\![.]\!]^\sharp \in \mathcal{I} \to (A^\sharp \times \mathit{Val}) \hookrightarrow (A^\sharp \times \mathit{Val})$*, taking as inputs an object and value, and returning an updated object and a value;* $\mathcal{P} \subseteq \mathit{Fields}$ *reserves private field identifiers for the concrete implementation of abstract methods in* $\mathcal{I}$*.*

Abstract objects in $A^\sharp$ are the possible values that an instance of a data structure can take. We use private fields to express the property of *interference freedom* from Herlihy and Shavit [6]. Namely, client code can only use public fields in $\mathit{Fields} \setminus \mathcal{P}$, and concrete implementations of abstract methods in $\mathcal{I}$ use private fields only.

$$\textsc{Acall}\ \frac{l(\mathtt{y}) = r \quad h^\sharp(r) = a \quad l(\mathtt{z}) = v \quad \mathtt{m'} \in \mathcal{I} \quad [\![\mathtt{m'}]\!]^\sharp(a, v) = (a', v')}{(\langle \mathtt{m}, \mathtt{x=\#\,y.m'(z)}, l, ls\rangle, (h^\sharp, h)) \xrightarrow{\tau} (\langle \mathtt{m}, \bullet, l[\mathtt{x} \mapsto v'], ls\rangle, (h^\sharp[r \mapsto a'], h))}$$

$$\textsc{Ccall}\ \frac{\begin{array}{c} l(\mathtt{y}) = r \quad l(\mathtt{z}) = v \quad l' = [\mathtt{m'.this} \mapsto r,\ \mathtt{m'.arg} \mapsto v] \\ ls' = \text{ if } \mathtt{m'} \in \mathcal{I} \text{ then } \mathsf{Before}(r, v) \text{ else } \mathsf{Nolin} \end{array}}{(\langle \mathtt{m}, \mathtt{x=y.m'(z)}, l, \mathsf{Nolin}\rangle, \sigma) \xrightarrow{\tau} (\langle \mathtt{m'}, \mathtt{m'}.comm, l', ls'\rangle, \sigma)}$$

$$\textsc{Lin}\ \frac{[\![\mathtt{b}]\!]l = true \quad h^\sharp(r) = a \quad \mathtt{m} \in \mathcal{I} \quad [\![\mathtt{m}]\!]^\sharp(a, v) = (a', v')}{(\langle \mathtt{m}, \mathtt{Lin(b)}, l, \mathsf{Before}(r, v)\rangle, (h^\sharp, h)) \xrightarrow{\tau} (\langle \mathtt{m}, \bullet, l[\mathtt{x} \mapsto v'], \mathsf{After}(r, v, v')\rangle, (h^\sharp[r \mapsto a'], h))}$$

$$\textsc{Intl}\ \frac{\gamma(t) = ts \quad (ts, \sigma) \xrightarrow{o} (ts', \sigma') \quad \forall t' \neq t,\ \neg inAtomic(\gamma(t'))}{(\gamma, \sigma) \xrightarrow{o} (\gamma[t \mapsto ts'], \sigma')}$$

Fig. 4: Semantics (excerpt)

Example 1 (Lock abstract data structure specification). In the spinlock example, we define $A^\sharp = \{\mathtt{Locked}, \mathtt{Unlocked}\}$, $\mathcal{I} = \{\mathtt{acquire}, \mathtt{release}\}$, and $\mathcal{P} = \{\mathtt{flag}\}$. We also define $[\![\mathtt{acquire}]\!]^\sharp(\mathtt{Unlocked}, v) = (\mathtt{Locked}, \mathit{Null})$ for any value v, and $[\![\mathtt{release}]\!]^\sharp(l, v) = (\mathtt{Unlocked}, \mathit{Null})$ for any input abstract lock l and value v.

Language Syntax The syntax of the language is detailed on Figure 3. In the sequel, we fix an abstract data structure specification $(A^\sharp, \mathcal{I}, [\![.]\!]^\sharp, \mathcal{P})$. The language provides constants (`n`, `null`, `true`), local variables (`x`,`y`,`z` ...), and arithmetic and boolean expressions (`e`, `b`). Regular commands (`c`) are standard, and common to the three languages. They include $\bullet$ (skip), an `assume(e)` statement, a `print(e)` instruction that emits the observable value of `e`, variable assignment of an expression, fields reads and updates, record allocation, non-deterministic choice (`+`), loops, and atomic blocks `atomic` $\langle c \rangle$. Concrete method calls are written `x=y.m(z)`.

Some instructions are specific to a language level. In $\mathcal{L}^\sharp$, abstract method calls on a abstract object are written `x =# y.m(z)`. For any `m` $\in \mathcal{I}$, such a call in a $\mathcal{L}^\sharp$ program is compiled to a concrete call `x=y.m(z)` in the $\mathcal{L}$ program. In $\mathcal{L}^\flat$, the `Lin(b)` instruction is used to annotate a linearization point.

Finally, a client program is defined by a map from method names in $\mathit{Methods} \setminus \mathcal{I}$ to their command, and a map from thread identifiers to their initial command. In the sequel, we will write `m`.*comm* for getting the command of method `m`, leaving the underlying program implicit.

Semantics For the sake of conciseness, we present here a partial view of the semantics, and refer the reader to the formal development [19] for full details[4].

[4] In our formal development, we use a continuation-based semantics to handle atomic blocks and method calls. This has proven to lighten the mechanisation of many proofs, by removing any recursivity from the small step semantics.

We assume a standard semantics $[\![\cdot]\!]$ for expressions, omitted here. Abstract objects are stored in an abstract heap, ranged over by $h^\sharp \in H^\sharp = \mathit{Ref} \to A^\sharp$. At the concrete level, abstract objects are implemented by regular, concrete objects, living in a concrete heap $h \in H = (\mathit{Ref} \times \mathit{Fields}) \to \mathit{Val}$. A shared memory, ranged over by $\sigma \in H^\sharp \times H$ is made of an abstract heap, and a concrete heap.

An intra-thread state $ts = \langle \mathtt{m}, \mathtt{c}, l, ls \rangle$ includes a current method $\mathtt{m}$, a current command $\mathtt{c}$, an environment $l \in \mathit{Lenv} = \mathit{Var} \to \mathit{Val}$, and a linearization state $ls \in \mathit{LinState}$, that we explain below. The intra-thread operational semantics, partially shown in the top three rules of Fig. 4, is a transition relation on intra-thread states. It is labelled with observable events (ranged over o, either a numeric value emitted by a `print`, or the silent event τ).

An abstract method call `x=# y.m(z)` is executed according to the abstract semantics $[\![\mathtt{m}]\!]^\sharp$, and modifies only the abstract heap (rule ACALL).

Concrete method calls (rule CCALL) behave as expected, but additionally manage the local linearization state. This linearization state notably keeps track of whether the execution of the current method is before its linearization point (Before) or not (After). Initially, the linearization state is set to Nolin. When control transfers to a method in $\mathcal{I}$ through a concrete method call, the linearization state changes from Nolin to Before (see rule CCALL). It switches to After when executing a `Lin` instruction (rule LIN), and then back to Nolin on method return. Linearization states are used in the simulation proof, and instrument $\mathcal{L}^\flat$ only.

At the $\mathcal{L}^\flat$ level, the `Lin` instruction also accounts for the effect on the abstract heap of concrete methods in $\mathcal{I}$: it performs the abstract atomic call $[\![\mathtt{m}]\!]^\sharp$ to the enclosing method $\mathtt{m}$, updating the local environment and abstract heap.

The interleaving of threads is handled in rule INTL, with relation $(\gamma, \sigma) \xrightarrow{o} (\gamma', \sigma')$ between global states (γ, σ), where γ maps thread identifiers to thread local states and σ is a shared memory. Mutual exclusion between atomic blocks is ensured by the $\neg \mathit{inAtomic}$ side condition.

Finally, program behaviors are defined on top of the interleaving semantics:

Definition 2 (Program behavior). *The observable behavior of a program p, written $\mathtt{obs}(p, \sigma_i)$, is either a finite trace of values emitted by a finite sequence of transitions or a infinite trace of values emitted by an infinite sequence of transitions from an initial shared memory σ_i.*

4 An RG Specification Entailing Semantic Refinement

In this section, we formalize our main result. We use the following notations and vocabulary. For a set A, an A predicate P is a subset of A. An element $a \in A$ satisfies the A predicate P, written $a \models P$, when $a \in P$. For two sets A and B, a relation R is an $A \times B$ predicate. We use infix notations for relations. State predicates are $(H^\sharp \times H \times \mathit{Lenv} \times \mathit{LinState})$ predicates, specifying shared memories and intra-thread states. A shared memory interference is a binary relation on $H^\sharp \times H$, and is used for relies and guarantees. We refer to both state predicates and shared memory interferences as *assertions*.

The rely-guarantee reasoning is done at the intermediate level $\mathcal{L}^\flat$, on instrumented programs, more precisely on the hybrid code of abstract methods implementations. Hence, assertions specify properties about the concrete and abstract heaps simultaneously.

Our work derives a compiler correctness result from a *generic* rely-guarantee specification. Of course, this cannot be achieved for an arbitrary RG specification, so we require some constraints on assertions. We present these now.

Semantic RG judgment A hybrid method $\mathtt{m}$ must be specified with a semantic RG judgment of the form $R, G, I \models_{\mathtt{m}} \{P\}\ c\ \{Q\}$, where P, Q, I are state predicates, R and G are shared memory interferences, and c is the body of method m. State predicate I is meant to specify the coherence invariant between abstract objects and their representation in the concrete heap. It is proved to be invariant separately (see Definition 6).

The RG judgment intuitively states that starting in a state satisfying P and invariant I, interleaving c with an environment behaving as prescribed by R (written $\rightarrow_R^*$), leads to a state satisfying Q. Additionally, this execution of c must be fully reflected by guaranty G. This intuition, typical of RG reasoning, is formalized by the first two conditions below.

Definition 3. *Judgment $R, G, I \models_{\mathtt{m}} \{P\}\ c\ \{Q\}$ holds whenever:*

1. *The post-condition is established from pre-condition and invariant:*
$$\left.\begin{array}{l}(\langle \mathtt{m}, \mathtt{c}, l, ls\rangle, \sigma) \rightarrow_R^* (\langle \mathtt{m}, \bullet, l', ls'\rangle, \sigma') \\ \wedge\ (l, ls, \sigma) \models P \cap I\end{array}\right\} \Rightarrow (l', ls', \sigma') \models Q$$
2. *Instructions comply with the guarantee:*
$$\left.\begin{array}{l}(\langle \mathtt{m}, \mathtt{c}, l, ls\rangle, \sigma) \rightarrow_R^* (\langle \mathtt{m}, \mathtt{c'}, l', ls'\rangle, \sigma') \rightarrow (\langle \mathtt{m}, \mathtt{c''}, l'', ls''\rangle, \sigma'') \\ \wedge\ (l, ls, \sigma) \models P \cap I\end{array}\right\} \Rightarrow \sigma'\ G\ \sigma''$$
3. *Linearization points are unique and non-blocking:*
$$\left.\begin{array}{l}(\langle \mathtt{m}, \mathtt{c}, l, ls\rangle, \sigma) \rightarrow_R^* (\langle \mathtt{m}, \mathtt{Lin}(b), l', ls'\rangle, (h^\sharp, h)) \\ \wedge\ (l, ls, \sigma) \models P \cap I \\ \wedge\ [\![b]\!] l' = true\end{array}\right\} \Rightarrow \begin{array}{l}\exists r, a, a', v, v',\ h^\sharp(r) = a \\ \quad \wedge\ [\![\mathtt{m}]\!]^\sharp(a, v) = (a', v') \\ \quad \wedge\ ls' = \mathsf{Before}(r, v)\end{array}$$

The third condition in Definition 3 is novel, and more subtle. It captures a necessary requirement to ensure that `Lin` instructions do not block programs ($[\![\mathtt{m}]\!]^\sharp$ is defined), and are unique (the linearization state is Before). This condition is essential to ensure that we can clean up the `Lin` instrumentation of hybrid programs, and that our semantic refinement is not vacuously true. We come back to this third condition in Section 6.

Specifying hybrid methods We now explain the specific RG judgment we require for hybrid methods.

Single Object Assertions. The above RG judgment involves state predicates and shared memory interferences. In fact, we build them from elementary bricks, *object predicates* and *object interferences*, that consider one object — one instance of a data structure — at a time, pointed to by a given reference.

Definition 4 (Object predicate, object interference). *Let $r \in Ref$. An* object predicate *P_r is a predicate on pairs of an abstract object and a concrete heap: $P_r \subseteq A^\sharp \times H$. An* object interference *R_r is a relation on pairs of an abstract object and a concrete heap: $R_r \subseteq (A^\sharp \times H) \times (A^\sharp \times H)$.*

Example 2 (Lock – Object invariant, object guarantees, and object relies).
The coherence invariant specifies that an abstract `Locked` (resp. `Unlocked`) lock is implemented in the concrete heap as an object whose field `flag` is set to `1` (resp. `0`). It is formalized as the following object predicate:

$$\mathit{ILock}_r \triangleq \{(\texttt{Locked}, h) \mid h(r, \texttt{flag}) = 1\} \cup \{(\texttt{Unlocked}, h) \mid h(r, \texttt{flag}) = 0\}$$

Object guarantees for `acquire` and `release` express the effect of the methods on the shared memory when called on a reference r. They are defined as:

$$G^r_{\texttt{rel}} \triangleq \{((a, h_1), (\texttt{Unlocked}, h_2)) \mid h_2 = h_1[r, \texttt{flag} \leftarrow 0]\}$$
$$G^r_{\texttt{acq}} \triangleq \{((\texttt{Unlocked}, h_1), (\texttt{Locked}, h_2)) \mid h_1(r, \texttt{flag}) = 0 \wedge h_2 = h_1[r, \texttt{flag} \leftarrow 1]\}$$

In $G^r_{\texttt{acq}}$, the assignment to `flag` is performed only if the `cas` succeeds.

Finally, both `acquire` and `release` have the same object rely, when called on a reference r – another thread could call both methods on the same reference. So we define the following object interference: for $\texttt{m} \in \{\texttt{rel}, \texttt{acq}\}$, $R^r_{\texttt{m}} \triangleq G^r_{\texttt{rel}} \cup G^r_{\texttt{acq}}$.

Lifting Single Object Assertions. We now need to lift object predicates and object interferences to state predicates and shared-memory interferences to enunciate the RG specifications of hybrid implementations. The challenge here is twofold: make the specification effort relatively light for the user, and, more importantly, sufficiently control the specifications so that we can derive our generic result.

An object predicate P_r is lifted to a state predicate by further specifying that, in the abstract heap, r points to an abstract object satisfying P_r:

$$\widehat{P_r} = \{(h^\sharp, h, l, ls) \mid \exists a, h^\sharp(r) = a \wedge \ (a, h) \models P_r\}$$

Similarly, for an object guarantee G_r, reference r should point to an abstract object in the abstract heap. Moreover, its effect on this object should be reflected in the resulting abstract heap. Formally:

$$\widehat{G_r} = \{((h^\sharp, h_1), (h^\sharp[r \mapsto a_2], h_2)) \mid \exists a_1, h^\sharp(r) = a_1 \wedge (a_1, h_1)\ G_r\ (a_2, h_2)\}$$

Lifting relies is a bit more subtle. When executing an hybrid implementation `m`, one should account for two kinds of concurrent effects: the client code, and the

rely of the method itself. To model the client code effect, we introduce a public shared memory interference, written R_c, that models any possible effect on the concrete heap, except modifying private fields in $\mathcal{P}$:

$$R_c = \{((h^\sharp, h_1), (h^\sharp, h_2)) \mid \forall r, f,\ f \in \mathcal{P} \Rightarrow h_1(r, f) = h_2(r, f)\}$$

As for the method's rely R_r, we should consider that it could occur on *any* abstract object present in the abstract heap. Hence, a lifted rely includes (i) the client public interference, and (ii) the method's rely R_r quantified over all r:

$$\widetilde{R} = R_c \cup \{((h_1^\sharp, h_1), (h_2^\sharp, h_2)) \mid \exists r, a_1, a_2, \\ h_1^\sharp(r) = a_1 \wedge (a_1, h_1)\ R_r\ (a_2, h_2) \wedge h_2^\sharp = h_1^\sharp[r \mapsto a_2]\}$$

The RG Specification. Before we define the RG proof obligation asked of hybrid method implementations, let us first recall the definition of stability.

Definition 5 (Stability). *State predicate P is* stable *w.r.t. shared memory interference R if $\forall l, ls, \sigma_1, \sigma_2$, $(\sigma_1, l, ls) \models P$ and $(\sigma_1 R\ \sigma_2)$ implies $(\sigma_2, l, ls) \models P$.*

Now, we fix an invariant I_r. For a method $\mathtt{m} \in \mathcal{I}$, let $G^r_{\mathtt{m}}$ and $R^r_{\mathtt{m}}$ be the object guaranty and rely of $\mathtt{m}$, as previously illustrated in Example 2. An RG specification for $\mathtt{m}$ includes an RG semantic judgment, and stability obligations:

Definition 6 (RG method specification). *The RG specification for method $\mathtt{m} \in \mathcal{I}$ includes the three following conditions:*

- *For all $r \in Ref$, $\widetilde{R}_{\mathtt{m}}, \widehat{G^r_{\mathtt{m}}}, \widehat{I}_r \models_{\mathtt{m}} \{\mathbb{P}_r\}\, \mathtt{m}.comm\, \{\mathbb{Q}_r\}$*
- *For all $r \in Ref$, predicate $\widehat{I}_r$ is stable w.r.t. $\widetilde{R}_{\mathtt{m}}$*
- *For all $r, r' \in Ref$, predicate $\widehat{I}_r$ is stable w.r.t. $\widehat{G^{r'}_{\mathtt{m}}}$*

In the above judgment, we impose the pre- and post-condition $\mathbb{P}_r$ and $\mathbb{Q}_r$. $\mathbb{P}_r$ expresses that (i) r points to an abstract object in the abstract heap, (ii) the linearization state is set to Before and (iii) the reserved local variable `this` of method $\mathtt{m}$ is set to r. $\mathbb{Q}_r$ expresses that (i) the linearization state is set to After, and (ii) the value virtually returned by the abstract method (when encountering the `Lin` instruction) matches the value returned by the concrete code. Intuitively, stability requirements ensure that $\widehat{I}_r$ is indeed an invariant of the whole program.

Main theorem So far, we have expressed requirements on hybrid methods, each taken in isolation. The last requirement we formulate is the consistency between relies and guarantees of methods. For a method $\mathtt{m}$, to ensure that $R_{\mathtt{m}}$ is indeed a correct over-approximation of its environment, we ask that $R_{\mathtt{m}}$ includes any guaranty $G_{\mathtt{m}'}$, where $\mathtt{m}'$ is a method that may be called concurrently to $\mathtt{m}$. This requirement is formalized by the following definition.

Definition 7 (RG consistency). *For all threads t, t' such that $t \neq t'$, all methods $\mathtt{m}, \mathtt{m'} \in \mathcal{I}$ and all $r \in Ref$, $\mathsf{is\_called}(t, \mathtt{m}) \wedge \mathsf{is\_called}(t', \mathtt{m'}) \Rightarrow G^r_{\mathtt{m}}, \subseteq R^r_{\mathtt{m}}$ where $\mathsf{is\_called}(t, \mathtt{m})$ indicates that $\mathtt{m}$ appears syntactically in the code of t.*

Relying on predicate is_called allows for accounting for data structures used according to an elementary protocol (such as single-pusher, single-reader buffers).

We finally package the formal requirements on hybrid implementations into the RGspec judgment and use it to state our main result, establishing that the target program semantically refines the source program.

Definition 8 (RGspec judgment). *Let $\mathcal{I} = \{\mathtt{m}_1 \ldots, \mathtt{m}_n\}$. $\mathcal{I}$ satisfies* RGspec*, written* RGspec$(\mathcal{I})$*, if $\forall\ i \in [1, n]$, an RG method specification is provided for $\mathtt{m}_i$, and RG consistency holds.*

Theorem 1 (Compiler correctness). *Let σ_i an initial shared memory satisfying the invariant I_r for all $r \in$ Ref allocated in it. If* RGspec$(\mathcal{I})$*, then $\forall p \in \mathcal{L}^\sharp, \mathtt{obs}(\mathtt{compile}\langle\mathcal{I}\rangle(p), \sigma_i) \subseteq \mathtt{obs}(p, \sigma_i)$.*

We insist that the client program p is arbitrary, modulo some basic syntactical well-formedness conditions (*e.g.*, no private field is accessed in the client code, or methods in $\mathcal{I}$ do not modify public fields).

Theorem 1 is phrased and proved *w.r.t.* an RG semantic judgment. In our formal development, we have developed a sound, syntax-directed proof system to discharge the RG semantic judgment, and have successfully used this system to prove the implementation of the spinlock, as well as the buffer data structure.

5 Case Study: Concurrent Buffers

This case study is taken from our larger verification project of a concurrent mark-and-sweep garbage collector [18]. Describing the full algorithm is out of the scope of this paper. Here, we only give an idea of why and how buffers are used.

In this algorithm, application threads, a.k.a *mutators*, are never blocked by the collector thread. They must therefore participate to the marking of potentially live objects. Buffers, so-called *mark buffers* in Domani et al.'s Java implementation [2], keep track of references to objects that are the roots of the graph of objects that may be live. Each thread, including the collector, owns a buffer. Only the owner of the buffer can push on it, and only the collector can read and pop from buffers.

In this section, we present abstract buffers and their fine-grained implementation. They are fully verified in our formal development [19].

An abstract buffer is a queue of bounded size *SIZE*, that we model by a list (of type `list value`). A buffer is pushed on one end of the list, and popped off from its other end. Buffers provide four methods: `isEmpty`, `top`, `pop`, and `push`. Due to space constraints, we focus here on methods `pop` and `push`. Their respective abstract semantics are[5]: $[\![\mathtt{pop}]\!]^\sharp(x :: b, v) = (b, \mathit{Null})$ and $[\![\mathtt{push}]\!]^\sharp(b, v) = (b^{+\!+}[v], \mathit{Null})$ if $|b| < \mathit{SIZE} - 1$.

The fine-grained implementation we prove is similar to that of Domani et al. [2], except that we use bounded-sized buffers. Buffers are objects with three fields (see Figure 5). Field `data` contains a reference to an array of fixed size *SIZE*, containing the elements of the buffer. Two other fields, `next_read` and

[5] The input value argument v is irrelevant for `pop`.

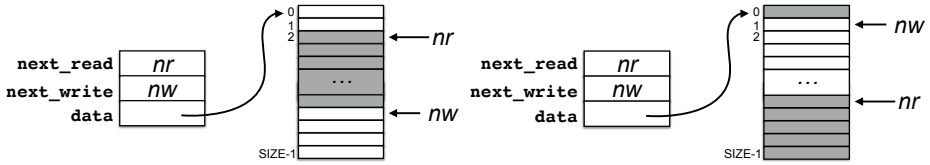


Fig. 5: Concrete buffers layout (examples). Elements contained in the buffer are colored in grey. Example on the right shows how the array is populated circularly.

`next_write`, indicate the bounds, within the array, of the effective content of the buffer. Field `next_read` contains the array index from which to read, while `next_write` contains the index of the first free slot in the array.

Pushing a value on a buffer consists in writing this value in the array, at position `next_write`, and then incrementing `next_write`. Conversely, popping a value from a buffer is done by incrementing `next_read`. In fact, the `data` array can be populated in a modulo fashion (see right example in Figure 5). The code for implementing buffers is given in Figure 6, and follows the above principles. Our core language does not include proper arrays, but we encode them with appropriate macros. The code is blocking when trying to pop on a empty buffer, or trying to push on a full buffer, as is the case for the abstract version. This is no limitation in practice: the size of buffers is chosen at initialization time, and can be upgraded at will.

```
def push(v) ::=
  nw=this.next_write
  nr=this.next_read
  d=this.data
  d[nw]=v
  nw=(nw+1) mod SIZE
  assume (nr<>nw) //no overflow
  atomic ⟨
    this.next_write=nw ; Lin(true)⟩
  return

def pop() ::=
  nr=this.next_read
  nw=this.next_write
  assume (nr<>nw) //no empty buffer
  nr=(nr+1) mod SIZE
  atomic ⟨
    this.next_read=nr ; Lin(true)⟩
  return
```

Fig. 6: Buffers in $\mathcal{L}^{\flat}$.

Using the approach illustrated on the lock in Section 4, we were able to formally establish RGspec($\{$`isEmpty`, `top`, `pop`, `push`$\}$), and then apply our generic refinement theorem, establishing the correctness of the compiler specialized to buffers. We refer to our formal development [19] for further details on the proof.

6 Establishing the Generic Simulation from RGspec

Theorem 1 is proved by establishing, from the RGspec($\mathcal{I}$) hypothesis, a simulation between the source program and its compilation. Here, we use the terminology of *backward* simulation, as is standard in compiler verification [11].

Definition 9 (Backward simulation). *A relation $\wr$ between states of $\mathcal{L}$ and $\mathcal{L}^\sharp$ is a backward simulation from $\mathcal{L}^\sharp$ to $\mathcal{L}$ if for any s_1, s_2, $s_1^\sharp$, $s_1^\sharp \wr s_1$ and $s_1 \xrightarrow{o} s_2$, implies that there exists $s_2^\sharp$ such that $s_1^\sharp \xrightarrow{o}{}^{*} s_2^\sharp$, and $s_2^\sharp \wr s_2$.*

Eliding customary constraints on initial and final states, such a simulation entails preservation of observable behaviors (Theorem 3 in [11]). We establish two backward simulations, from $\mathcal{L}$ to $\mathcal{L}^\flat$ and from $\mathcal{L}^\flat$ to $\mathcal{L}^\sharp$, which we compose.

Leveraging RGspec($\mathcal{I}$). The key point is to carry, within the simulation relation, enough information to leverage $\mathsf{RGspec}(\mathcal{I})$. This is necessary for both simulations, so we factorize the work by expressing a rich semantic invariant $\mathbb{I}$ over the execution of the $\mathcal{L}^\flat$ program. To simplify its definition and its proof, $\mathbb{I}$ is built as a combination of thread-local invariants.

Thread local invariant. For a thread t, the invariant $\mathbb{I}_t$ includes three kinds of information. First, it ensures various well-formedness properties of intra-thread states. Second, it demands that $\widehat{I}_r$, the coherence invariant, holds for all r. The third information is more subtle. When executing a hybrid method $\mathtt{m}$ called on a reference r, to leverage its specification $\widetilde{R}_\mathtt{m}, \widehat{G^r_\mathtt{m}}, \widehat{I}_r \models_\mathtt{m} \{\mathbb{P}_r\}\, \mathtt{m}.comm\, \{\mathbb{Q}_r\}$, we keep track, in $\mathbb{I}_t$, that the state is reachable from a state satisfying $\widehat{I}_r$ and $\mathbb{P}_r$. Recall that Definition 3 uses the abstract semantics $\rightarrow_{\widetilde{R}_\mathtt{m}}$. When defining $\mathbb{I}_t$, we generalize $\rightarrow_{\widetilde{R}_\mathtt{m}}$ into relation $\rightarrow_{R_t}$, where $R_t \triangleq \bigcap_{\mathtt{m} \in \mathsf{is\_called}(t)} \widetilde{R}_\mathtt{m}$. Rely R_t overapproximates interferences of threads concurrent to t, while being precise enough to deal with any method $\mathtt{m}$ called by t, since $R_t \subseteq \widetilde{R}_\mathtt{m}$.

Global invariant. We define $\mathbb{I} \triangleq \{(\gamma, \sigma) \mid \forall t, (\gamma(t), \sigma) \models \mathbb{I}_t\}$. To prove that $\mathbb{I}$ holds on the interleaving semantics, we first prove that $\mathbb{I}_t$ are preserved by the intra-thread steps. Besides, we prove their preservation by other threads' steps:

Lemma 1. *Let γ, σ and $t \neq t'$ be such that $(\gamma(t), \sigma) \models \mathbb{I}_t$ and $(\gamma(t'), \sigma) \models \mathbb{I}_{t'}$. If $(\gamma(t'), \sigma) \xrightarrow{o} (ts', \sigma')$, then $(\gamma(t), \sigma') \models \mathbb{I}_t$.*

Simulation relations. For both compilation phases, we build an intra-thread, or *local*, simulation that we then lift at the inter-thread level. Both relations are defined using the same pattern: in a pair of related states, the $\mathcal{L}^\flat$ state satisfies $\mathbb{I}_t$. It remains to encode in the relation the matching between execution states.

Local high simulation. For the first compilation phase, a whole execution of an hybrid method is simulated by a single abstract step, occurring at the linearization point. We therefore build a *1-to-0/1* backward simulation.

Relation $\wr^\sharp_\flat$ states that shared memories are equal on the domains of heaps in $\mathcal{L}^\sharp$. Local environments are trickier to relate. In client code, they simply are equal. During a hybrid method call `x=y.m(z)`, before the `Lin` point, the abstract environment is equal to the environment of the $\mathcal{L}^\flat$ caller. After the `Lin` point, the only mismatch is on variable `x`, which has been updated in $\mathcal{L}^\sharp$, but not yet in $\mathcal{L}^\flat$.

Proving that $\wr^\sharp_\flat$ is indeed a simulation follows the above three phases. Steps by client code are matched *1-to-1*; inside a hybrid method, steps match *1-to-0* until the `Lin` point; the `Lin` step is matched *1-to-1*; after the `Lin` point, steps match *1-to-0* until the return instruction. At method call return, we use $\mathsf{RGspec}(\mathcal{I})$ via $\mathbb{I}_t$, and in particular $\mathbb{Q}_r$, to prove that environments coincide on `x` again.

Local low simulation. When simulating from $\mathcal{L}$ to $\mathcal{L}^\flat$, `Lin` instructions have been replaced by a $\bullet$. It is therefore a *1-to-1* backward simulation. Recall that $\mathcal{L}$ semantics contains no *LinState* nor abstract heap. Relation $\wr^\flat$ therefore only states the equality of local environments and concrete heaps.

Proving this simulation is what makes the third item in Definition 3 necessary. Indeed, we can match the $\bullet$ step in the $\mathcal{L}$ program only if the `Lin` instruction in the $\mathcal{L}^\flat$ program is non-blocking.

Global simulations. The independent proof of the invariant simplifies the lifting of simulations. Indeed, except for the part about $\mathbb{I}_t$, that is already proved invariant by other threads' steps, relations keep track of the same information for all threads. Hence, their preservation by the interleaving essentially comes for free.

7 Related Work and Conclusion

Related Work The literature on linearizability verification is vast. Dongol *et al.* [3] provide a comprehensive survey of techniques for verifying linearizability *w.r.t.* the seminal definition of Herlihy [7]. Notably, a number of works use concurrent program logics, most of them influenced by Jones's rely-guarantee [9] and O'Hearn's Concurrent Separation Logic [14]. Both ideas have been combined into logics like RGSep [17], SAGL [4], and more recently Iris [10]. Another logic is worth mentioning, although not directly applied to linearizability: Sergey *et al.* [15] provide a Coq framework to mechanically verify fine-grained concurrent algorithms, based on the FCSL logic. FCSL's soundness is formally proved in Coq *w.r.t.* a denotational semantics, but the shallow embedding of programs in Coq makes the approach hard to use in compiler verification. While the above logics have highly expressive powers, they are not formally linked with observational refinement, which is what we aim at. Filipović *et al.* [5] characterize linearizability in terms of observable refinement, on top of a non-operational semantics. Here, we want to express our main result in terms of observable refinement directly.

Our work is inspired by the technique outlined in [16]. Here, we use a simple rely-guarantee formulation: we think it is a good balance between the logic expressivity and its mechanization effort. Indeed, it is enough to prove the buffers used in the concurrent garbage collector we verify in a related project [18].

The technique presented in [16] lacks a mechanized soundness proof that would be suitable to a verified compiler infrastructure. Our work provides such a foundation. Liang *et al.* [13] tackle a problem similar to ours. They define a simulation parameterized by relies and guarantees, and compositionality rules, to reason about program transformations. In [12], they combine it with the technique in [16] to verify linearizability. This work is not mechanized.

The work of Derrick *et al.* [1], formalized in the KIV tool, is also closely related. Like us, they express linearizability through thread-local proof obligations, establishing systematically inter-thread simulations. Our work differs in the nature of the proof obligations: we choose to express them in terms of rely-guarantee, so we can discharge them using program logics.

Jagannathan *et al.* [8] propose an atomicity refinement methodology for verified compilation. Their final theorem, mechanized in Coq, is expressed as a behavior preservation, but the proof methodology is completely different from the one presented here. They provide compositional rules to symbolically refine high-level atomic blocks into fine-grained low-level code.

Conclusion This work embeds the notion of linearizable data structures in a compiler formally verified in Coq. As such, this represents a mechanized soundness foundation for Vafeiadis' technique [16], phrased in terms of semantic refinement. To achieve this result, we establish, starting from proof obligations expressed in terms of rely-guarantee reasoning, a generic backward simulation theorem. We use our meta-theorem to compile two fine-grained concurrent data structures: an illustrative spinlock and a realistic implementation of concurrent buffers. The development is $13kloc$ long, an effort of one man-year.

This work is part of a larger project aiming at verifying concurrent compilation mechanisms such as garbage collectors or dynamic allocators, and plugging them in a verified compiler. In an ongoing work, we are in particular connecting the present work with a soundness proof of a concurrent garbage collector [18]. Indeed, the language we designed to implement the collector is enhanced with constructs facilitating the programming and the proof of compiler services, such as introspection on objects and high-level management of threads roots. Notably, the collector and mutators share abstract concurrent buffers to keep track of potentially live objects. The work presented here will allow to propagate the formal soundness of the collector down to its low-level, fine-grained implementation.

References

1. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, January 2011.
2. T. Domani, E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, Y. Levanoni, E. Petrank, and I. Yanover. Implementing an on-the-fly garbage collector for Java. In *Proc. of ISMM'00*, 2000.
3. B. Dongol and J. Derrick. Verifying linearisability: A comparative survey. *ACM Comput. Surv.*, September 2015.
4. X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.
5. I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *ESOP*, 2009.
6. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
7. M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 1990.
8. S. Jagannathan, V. Laporte, G. Petri, D. Pichardie, and J. Vitek. Atomicity refinement for verified compilation. *TOPLAS*, 2014.
9. C. B. Jones. Specification and design of (parallel) programs. In *IFIP*, 1983.

10. R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, 2015.
11. X. Leroy. A formally verified compiler back-end. *JAR*, 2009.
12. H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, 2013.
13. H. Liang, X. Feng, and M. Fu. Rely-Guarantee-based simulation for compositional verification of concurrent program transformations. *TOPLAS*, 2014.
14. P. W. O'Hearn. Resources, concurrency, and local reasoning. *TCS*, 2007.
15. I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, 2015.
16. V. Vafeiadis. *Modular Fine-Grained Concurrency Verification*. PhD thesis, University of Cambridge, July 2007.
17. V. Vafeiadis and M. J. Parkinson. A marriage of Rely/Guarantee and separation logic. In *CONCUR*, 2007.
18. Y. Zakowski, D. Cachera, D. Demange, G. Petri, D. Pichardie, S. Jagannathan, and J. Vitek. Verifying a concurrent garbage collector using a rely-guarantee methodology. *ITP*, 2017. Accepted for publication.
19. Y. Zakowski, D. Cachera, D. Demange, and D. Pichardie. Companion website. Available at `http://www.irisa.fr/celtique/ext/simulin`.